

Auto-generating Test Sequences using Model Checkers: A Case Study

Mats P. E. Heimdahl¹, Sanjai Rayadurgam¹, Willem Visser², Devaraj George¹, and Jimin Gao¹

¹Department of Computer Science and Engineering, University of Minnesota

² NASA Ames Research Center

Abstract

Use of model-checking approaches for test generation from requirement models have been proposed by several researchers. These approaches leverage the witness (or counter-example) generation capability of model-checkers for constructing test cases. Test criteria are expressed as temporal properties. Witness traces generated for these properties are instantiated to create complete test sequences, satisfying the criteria. State-space explosion can, however, adversely impact model-checking and hence such test generation. Thus, there is a need to validate these approaches against realistic industrial sized system models to learn how well these approaches scale. To this end, we conducted a case study using six models of progressively increasing complexity of the mode-logic in a flight-guidance system, written in the RSML^{-e} language. We developed a framework for specification-based test generation using the NuSMV model-checker and code based test case generation using Java Pathfinder, and collected time and resource usage data for generating test cases using symbolic, bounded, and explicit state model-checking algorithms. This paper briefly discusses the approach, presents the results from the study and analyzes its implications.

1. Introduction

Software development for high assurance systems, such as the software controlling aeronautics applications and medical devices, is a costly and time consuming process. In such projects, the validation and verification phase (V&V) consume approximately 50%–70% of the software development resources. Thus, automatic generation of test cases from requirement specifications has found considerable interest in the research community. Such automation could result in dramatic time and cost savings, especially for verifying safety-critical systems.

Model checking techniques have been proposed as one method of achieving this automation [3, 9, 2, 10, 20, 16]. These proposed test case generation approaches leverage

the witness (or counter-example) generation capability of model-checkers for constructing test cases. Test criteria are expressed as temporal properties. Witness traces generated for these properties are instantiated to create complete test sequences, satisfying the criteria. Nevertheless, one of the issues that often stymies model-checking is the state-space explosion problem. As the size of the state-space to be explored increases, model-checking might become too time-consuming or infeasible. But in the context of test generation based on structural properties, one is interested in falsifying properties so that counter-examples can be instantiated to test sequences. We have hypothesized that finding violations of the properties characterizing a test case is easy and that the counter-examples can be constructed easily even for large models.

While these ideas are appealing there is a need to validate the approach using realistic models of critical systems. To this end, we conducted a case study using six models of progressively increasing complexity of the mode-logic in a flight-guidance system, written in the RSML^{-e} language [22, 23]. We developed a framework for specification-based test generation using the NuSMV [19] model-checker and code based test case generation using Java Pathfinder [25] and collected time and resource usage data for generating test cases using symbolic, bounded, and explicit state model-checking algorithms. The purpose of this study was to determine if a model-checking based approach to test generation could scale to software system models of industrial size and complexity. Further, we were also interested in applying a bounded search of the state-space and see if this improved the performance without adversely affecting the test generation capability.

To summarize our findings, our case study points out limitations of symbolic as well as explicit state model checkers when used for test case generation. A bounded model checker, however, performed very well in our application domain and shows tremendous promise.

The rest of the paper is organized as follows. Section 2 provides a short overview of related efforts in the area of test-generation using model checking techniques and briefly describes our overall approach. We describe how we con-

ducted our case study in Section 3, and present the FGS case example in Section 4. Sections 5 and 6 briefly discuss RSML^{-e} and the test coverage criteria used for this study. Section 7 analyzes the results obtained and finally Section 9 discusses the implications of the results and points to future studies and experiments that are further required to validate model-checking based test generation approaches.

2. Finding Tests with a Model Checker

Model checkers build a finite state transition system and exhaustively explore the reachable state space searching for violations of the properties under investigation [8]. Should a property violation be detected, the model checker will produce a counter-example illustrating how this violation can take place. In short, a counter-example is a sequence of inputs that will take the finite state model from its initial state to a state where the violation occurs.

A model checker can be used to find test cases by formulating a test criterion as a verification condition for the model checker. For example, we may want to test a transition (guarded with condition C) between states A and B in the formal model. We can formulate a condition describing a test case testing this transition—the sequence of inputs must take the model to state A ; in state A , C must be true, and the next state must be B . This is a property expressible in the logics used in common model checkers, for example, the logic LTL. We can now challenge the model checker to find a way of getting to such a state by negating the property (saying that we assert that there is no such input sequence) and start verification. The model checker will now search for a counterexample demonstrating that this property is, in fact, satisfiable; such a counterexample constitutes a test case that will exercise the transition of interest. By repeating this process for each transition in the formal model, we use the model checker to automatically derive test sequences that will give us transition coverage of the model. The proposed test generation process is outlined in Figure 1. Naturally, the same thinking can be applied to the generation of test cases from source code, for example, from Java as we will illustrate later in the paper.

Several research groups are actively pursuing model checking techniques as a means for test case generation.

Gargantini and Heitmeyer [10] describe a method for generating test sequences from requirements specified in the SCR notation. To derive a test sequence, a *trap property* is defined which violates some known property of the specification. In their work, they define trap properties that exercise each case in the event and condition tables available in SCR—this provides a notion of branch coverage of an SCR specification.

Ammann and Black [2, 1] combine mutation analysis with model-checking based test case generation. They de-

fine a specification based coverage metric for test suites using the ratio of the number of mutants killed by the test suite to the total number of mutants. Their test generation approach uses a model-checker to generate mutation adequate test suites. The mutants are produced by systematically applying mutation operators to both the properties specifications and the operational specification, producing respectively, both positive test cases which a correct implementation should pass, and negative test cases which a correct implementation should fail.

Rayadurgam, *et al.* in [20] provide a formalism suitable for structural test-case generation using model checkers and in [21] illustrate how this approach can be applied to a formal specification language. They also presented a framework for specification centered testing in [13].

Lee, *et al.* [16] formulate a theoretical framework for using temporal logic to specify data flow test coverage criteria. They also discuss various techniques for reducing the size of the test set generated by the model checker [15]. The underlying argument in all these works, as in our own earlier work, is that when test criteria can be appropriately formulated as temporal logic formulas, one could use model-checking to produce witnesses for those formulas, which could then be seen as test sequences satisfying the coverage criteria.

However, to our knowledge, not much experimental data is available about the efficacy of model-checking based test-generation for realistic systems. Our goal is to conduct a series of studies using realistic systems, apply the techniques and examine how well these techniques perform and to what extent they scale up.

3. Case Study Overview

In our case study, we were interested in answering four questions:

1. If we naively generate one test case for each structure we want to cover, how many test cases will be generated for various coverage criteria?
2. Does test case generation using symbolic and bounded model checking scale to realistic systems?
3. Where do the test case generation capabilities of symbolic and bounded model checking break down?
4. Can a code model checker, such as JPF, be used to find test cases based on realistic code?

To answer these questions, we devised a rigorous case study evaluating the test case generation capabilities of model checkers. We have developed a test case generation engine integrated in our NIMBUS toolset for the development of RSML^{-e} specifications [22] (described in Section 5). This test case generator allows us to generate test cases to various structural coverage criteria using the NuSMV model

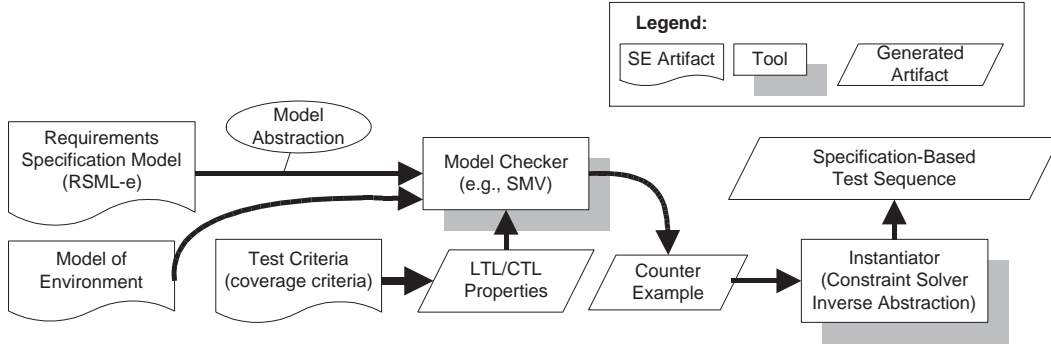


Figure 1. Test sequence generation overview and architecture.

checker. The coverage criteria we have used in this study are discussed in Section 6.

To stress the capabilities of NuSMV, we wanted to work with models with realistic structure as well as realistic size. In a related project, Rockwell Collins Inc., in collaboration with the University of Minnesota, have developed a collection of progressively more complex RSML^{-e} models of the mode logic of a flight guidance system (FGS). The models range from a very simple “toy-version” of the FGS (FGS00) to a close to production version of the logic (FGS05). The case example is discussed in some detail in Section 4.

We performed the case study by conducting the following steps:

1. Use NIMBUS to automatically generate LTL trap properties for various coverage criteria for FGS00 through FGS05.
2. Use the symbolic as well as bounded model checkers provided in NuSMV to generate counterexamples for the suites of trap properties.
3. Automatically process the counterexamples to provide test cases suitable for use in a test automation environment.

During the case study, we collected information on (1) how many test cases were generated, (2) run time and memory usage of the model checkers, and (3) the average length of the test cases generated.

To complete the case study, we investigated the feasibility of using a code model checker to complete the test suites derived from the formal specification. This capability would be used should the specification-based tests not provide adequate coverage of the implementation. To this end, we derived Java code from the formal specifications, executed a test suite generated from the specification, identified branches that were not covered, and derived tests for these branches using Java Pathfinder Java model checker [25].

In the remainder of this paper we provide a detailed description of the artifacts and activities involved in the case study.

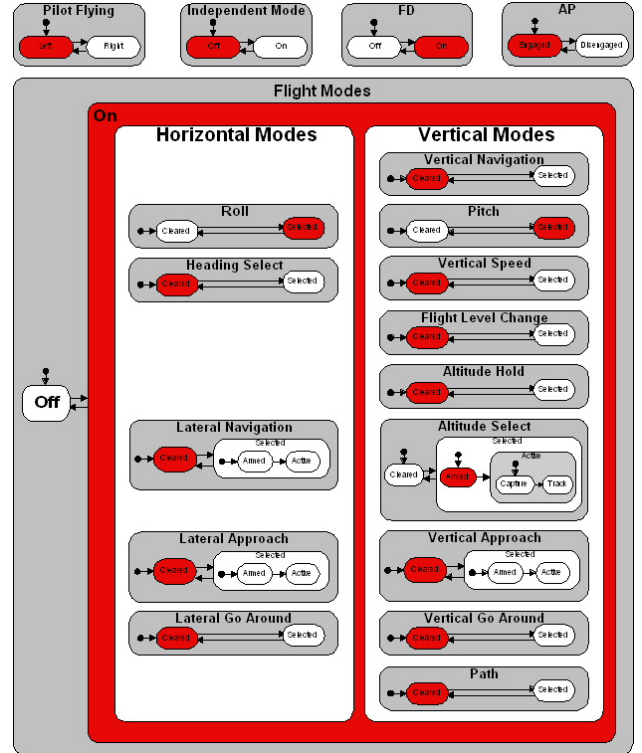


Figure 2. Flight Guidance System

4. Flight Guidance System

A Flight Guidance System (FGS) is a component of the overall Flight Control System (FCS) in a commercial aircraft. It compares the measured state of an aircraft (position, speed, and altitude) to the desired state and generate pitch and roll guidance commands to minimize the difference between the measured and desired state¹. The FGS can be

¹ We thank Dr. Steve Miller and Dr. Alan Tribble of Rockwell Collins Inc. for the information on flight control systems and for letting us use the RSML^{-e} models they have developed using NIMBUS.

broken down to mode logic, which determines which lateral and vertical modes of operation are active and armed at any given time, and the flight control laws that accept information about the aircraft's current and desired state and compute the pitch and roll guidance commands. In this case study we have used the mode logic.

Figure 2 illustrates a graphical view of a FGS in the NIMBUS environment. The primary modes of interest in the FGS are the horizontal and vertical modes. The horizontal modes control the behavior of the aircraft about the longitudinal, or roll, axis, while the vertical modes control the behavior of the aircraft about the vertical, or pitch, axis. In addition, there are a number of auxiliary modes, such as half-bank mode, that control other aspects of the aircraft's behavior.

The FGS is ideally suited for test case generation using model checkers since it is discrete—the mode logic consists entirely of enumerated and Boolean variables. As mentioned earlier, we used six models that are of progressively increasing complexity. An indication of the model size can be found in Table 1.

5. NIMBUS and RSML^{-e}

Figure 3 shows an overview of the NIMBUS tools framework we have used as a basis for our test case generation engine. The user builds a behavioral model of the system in the fully formal and executable specification language RSML^{-e} (see below). After evaluating the functionality and behavioral correctness of the specification using the NIMBUS simulator, users can translate the specifications to the PVS or NuSMV input languages for verification (or test case generation as is the case in this report). The set of LTL trap properties required to use NuSMV to generate test sequences are obtained by traversing the abstract syntax tree in NIMBUS and then outputting sets of properties whose counterexamples will provide the correct coverage (the coverage criteria an associated properties are discussed in the next section).

To generate test cases in NIMBUS, the user would invoke the following steps:

Model creation and trap property generation: The formal model in NuSMV can be generated automatically from the RSML^{-e} specification from the NIMBUS command line. The test criterion is specified as a command line argument when building the NuSMV model. The result of this command is an SMV model of the system and a collection of trap properties whose counterexamples will provide the desired coverage.

Counterexample generation using NuSMV: The model and the trap properties are merged and given to the NuSMV tool. A Unix script invokes the NuSMV tool

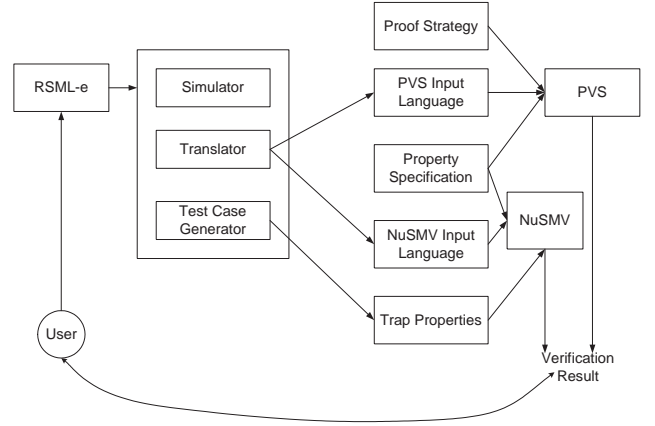


Figure 3. Verification Framework.

in interactive mode, reads the model, flattens the hierarchy, encodes the variables, and checks the specifications for the trap properties. After completing the script, we have collected the counter example traces for all trap properties in a text file.

Concrete test case generation from NuSMV Output:

For any counterexample, the trace information from NuSMV contains only delta changes in each subsequent state following the initial state. Therefore, to generate test sequences, we need to remember the value of the variables in the initial state configuration so that we can construct usable test cases by applying the delta changes to the initial configuration. The processing of the counterexamples and generation of an intermediate test representation is currently achieved with a simple piece of software implemented in C. The intermediate test representation contains (1) the input in each step, (2) the expected state changes (to state variables internal to the RSML^{-e} model), and (3) the expected outputs (if any).

The NIMBUS tools discussed above all operate on the RSML^{-e} notation—RSML^{-e} is based on the Statecharts [12] like language Requirements State Machine Language (RSML) [18]. RSML^{-e} is a fully formal and synchronous data-flow language without any internal broadcast events (the absence of events is indicated by the ^{-e}).

An RSML^{-e} specification consists of a collection of input variables, state variables, input/output interfaces, functions, macros, and constants; *input variables* are used to record the values observed in the environment, *state variables* are organized in a hierarchical fashion and are used to model various states of the control model, *interfaces* act as communication gateways to the external environment, and *functions and macros* encapsulate computations providing increased readability and ease of use.

Figure 4 shows a specification fragment of an RSML^{-e}

	FGS00		FGS01		FGS02		FGS03		FGS04		FGS05	
	RSML	SMV	RSML	SMV	RSML	SMV	RSML	SMV	RSML	SMV	RSML	SMV
#lines of code	287	423	455	639	774	1045	1034	1186	1781	2052	2564	2902
#vars/BDD's	19	109	27	167	43	281	57	353	90	615	142	849

Table 1. Data on the size of the RSML^{-e} and SMV FGS models.

```

STATE_VARIABLE ROLL : Base_State
  PARENT           : Modes.On
  INITIAL_VALUE    : UNDEFINED
  CLASSIFICATION   : State

  TRANSITION UNDEFINED TO Cleared IF NOT Select_ROLL()
  TRANSITION UNDEFINED TO Selected IF Select_ROLL()
  TRANSITION Cleared TO Selected IF Select_ROLL()
  TRANSITION Selected TO Cleared IF Deselect_ROLL()

END STATE_VARIABLE

MACRO Select_ROLL() :
  TABLE
    Is_No_Nonbasic_Lateral_Mode_Active() : T;
    Modes = On                           : T;
  END TABLE
END MACRO

MACRO Deselect_ROLL() :
  TABLE
    When_Nonbasic_Lateral_Mode_Activated() : T *;
    When(Modes = Off)                       : * T;
  END TABLE
END MACRO

```

Figure 4. A small portion of the FGS specification in RSML^{-e}.

specification of the Flight Guidance System². The figure shows the definition of a state variable, ROLL. ROLL is the default lateral mode in the FGS mode logic.

The conditions under which the state variable changes value are defined in the TRANSITION clauses in the definition. The condition tables are encoded in the macros, Select_ROLL and Deselect_ROLL. The use of macros not only improve the readability of the specifications but also help localize errors and future changes. The tables are adopted from the original RSML notation—each column of truth values represents a conjunction of the propositions in the leftmost column (a “*” represents a “don’t care” condition). If a table contains several columns, we take the disjunction of the columns; thus, the table is a way of expressing conditions in a disjunctive normal form.

6. Coverage Criteria

For the case study described in this report, we have selected to use three representative specification coverage criteria; state coverage, decision coverage (in the RSML^{-e}

²We use here the ASCII version of RSML^{-e} since it is much more compact than the more readable typeset version.

context called table coverage), and a version of MC/DC coverage [4] called clause-wise condition coverage.

In the following discussion, a *test case* is to be understood as a sequence of values for the input variables in an RSML^{-e} specification. This sequence of inputs will guide the RSML^{-e} specification from its initial state to the structural element, for example, a transition, the test cases was designed to cover. A *test suite* is simply a set of such test cases. As we briefly explained, trap properties are used to generate counter-examples using a model checker. These properties are derived from the structural coverage criteria. For the purposes of illustration, we use the FGS example discussed in Section 4.

State coverage:

Definition 1. A test suite is said to achieve state coverage of a state variable in an RSML^{-e} specification, if for each possible value of the state variable there is at least one test case in the test suite that assigns that value to the given variable. The test suite achieves state coverage of the specification if it achieves state coverage for each state variable.

Consider, for example, the state variable ROLL in the FGS specification example:

```
STATE_VARIABLE ROLL : { Cleared, Selected, UNDEFINED };
```

A test suite would achieve state coverage on ROLL, if for each of its three different possible values, there is a test case in which ROLL takes that value. Note that a single test case might actually achieve this coverage by assigning different values to ROLL at different points in the sequence. To provide a comprehensive test suite, however, in this case study we generate one test case for each state variable value. One could use the following LTL formulas to generate the test cases:

1. $G \sim (ROLL = Cleared)$
2. $G \sim (ROLL = Selected)$
3. $G \sim (ROLL = UNDEFINED)$

In each case, the property asserts that ROLL can never have a specific value and the counter-example produced is a sequence of values for the system variables starting from an initial state and ending in a state where ROLL has the specific value.

Decision coverage (table coverage):

Definition 2. A test suite is said to achieve decision coverage of a given $RSML^{-e}$ specification, if each guard condition (specified as either an AND/OR table or as a standard Boolean expression) evaluates to true at some point in some test case and evaluates to false at some point in some other test case in the test suite.

We also refer to this coverage criterion as table coverage since AND/OR tables typically are used for every decision in $RSML^{-e}$. As an example, consider the transition defined for the ROLL state variable in Figure 4.

If we consider the transition from Cleared to Selected guarded by the condition encapsulated in the `Select_ROLL()`, test cases to provide decision coverage of this transition can be generated using the following two trap properties.

1. $G((\text{Select_ROLL}()) \rightarrow \sim(\text{ROLL} = \text{Selected}))$
2. $G(\sim(\text{Select_ROLL}()) \rightarrow (\text{ROLL} = \text{Selected}))$

Clause-wise transition coverage: Finally, to exercise the approach with a complex test coverage criterion, we look at the code based coverage criterion called modified condition/decision coverage (MC/DC) and define a similar criterion. MC/DC was developed to meet the need for extensive testing of complex boolean expressions in safety-critical applications [4]. Ideally, one should test every possible combination of values for the conditions, thus achieving *compound condition coverage*. Nevertheless, the number of test cases required to achieve this grows exponentially with the number of conditions and hence becomes huge or impractical for systems with tens of conditions per decision point. MC/DC was developed as a practical and reasonable compromise between decision coverage and compound condition coverage. It has been in use for several years in the commercial avionics industry. A test suite is said to satisfy MC/DC if executing the test cases in the test suite will guarantee that:

- every point of entry and exit in the program has been invoked at least once,
- every basic condition in a decision in the program has taken on all possible outcomes at least once, and
- each basic condition has been shown to independently affect the decision's outcome

where a basic condition is an atomic Boolean valued expression that cannot be broken into Boolean sub-expressions. A basic condition is shown to independently affect a decision's outcome by varying only that condition while holding all other conditions at that decision point fixed. Thus, a pair of test cases must exist for each basic condition in the test-suite to satisfy MC/DC. However, test case pairs for different basic conditions need not necessarily be disjoint. In

fact, the size of MC/DC adequate test-suite can be as small as $N + 1$ for a decision point with N conditions.

If we think of the system as a realization of the specified transition relation, it evaluates each guard on each transition to determine which transitions are enabled and thus each guard becomes a decision point. The predicates in turn are constructed from clauses—the basic conditions.

Definition 3. A test suite is said to achieve clause-wise transition coverage (CTC) for a given transition of a variable in an $RSML^{-e}$ specification, if every basic Boolean condition in the transition guard is shown to independently affect the transition.

Consider the following transition example adopted from an avionics system related to the FGS:

```
EQUALS PowerOn IF
TABLE
PREV_STEP(DOI) IN_STATE AttemptingOn      : F T;
PREV_STEP(DOI) IN_ONE_OF {PowerOff, Unknown}: T F;
DOIStatus = On                             : T T;
AltitudeStatus IN_STATE Below              : T *;
ivReset                                    : F F;
END TABLE
```

To show that each of the basic conditions in the rows independently affects the transition, one should produce a set of test cases in which for any given basic condition there are two test cases, such that one makes the basic condition *true* and the other makes it *false*, the rest of the basic conditions have the same truth values in both test cases, and in one test case the transition is taken while in the other it is not. For the purposes of this example, let us just consider the first column. We may generate the trap properties by examining the truth value for each row in the first column as follows:

0. $G((\sim R1 \ \& \ R2 \ \& \ R3 \ \& \ R4 \ \& \ \sim R5) \rightarrow \sim(\text{POST}));$
1. $G((R1 \ \& \ R2 \ \& \ R3 \ \& \ R4 \ \& \ \sim R5) \rightarrow \text{POST});$
2. $G((\sim R1 \ \& \ \sim R2 \ \& \ R3 \ \& \ R4 \ \& \ \sim R5) \rightarrow \text{POST});$
3. $G((\sim R1 \ \& \ R2 \ \& \ \sim R3 \ \& \ R4 \ \& \ \sim R5) \rightarrow \text{POST});$
4. $G((\sim R1 \ \& \ R2 \ \& \ R3 \ \& \ \sim R4 \ \& \ \sim R5) \rightarrow \text{POST});$
5. $G((\sim R1 \ \& \ R2 \ \& \ R3 \ \& \ R4 \ \& \ R5) \rightarrow \text{POST});$

where R_i stands for the basic condition in the i^{th} row of the table and POST represents the post-state condition $DOI = \text{PowerOn}$.

MC/DC test cases come in pairs, one where the atomic condition evaluates to false and one where it evaluates to true, but no other atomic conditions in the Boolean expression are changed. In the example above, trap properties 0 and 1 provide coverage of $R1$. Unfortunately, the model checking approach to test case generation is incapable of capturing such constraints over two test sequences. To work around this problem, we have developed a novel alternative that leverages a model checker for complete and accurate MC/DC test case generation. We automatically rewrite the system model by introducing a small number of auxiliary variables to capture the constraints that span more

	FGS00	FGS01	FGS02	FGS03	FGS04	FGS05
State Coverage	33	54	78	99	117	246
Table Coverage	40	68	98	136	196	342
MCDC Coverage	32	54	77	118	205	323

Table 2. Trap Properties generated per test criterion for all FGS Models

than one test-sequence. We also introduce a special system reset transition to restore a system to its initial state. With these small modifications, a test constraint spanning two sequences in the original model can be expressed as a constraint on a single test-sequence in the modified model. Model-checking techniques can then be employed to generate this single test sequence which can be later factored into two separate test-sequences for the original model satisfying the actual test criteria. This process has been fully automated, and used to generate the MC/DC like tests in this case study.

To summarize, we have automated the generations of trap properties for a collection of structural coverage criteria of formal specifications. In this case study we are using the three representative criteria described above; state coverage, decision coverage, and clause-wise condition coverage. The experiential results of using model checkers to generate test suites to these coverage criteria are presented next.

7. Experimental Results and Discussion

The results of our case study are presented in Tables 2 through 4. Table 2 provides a count of the number of trap properties generated for each FGS model for each coverage criterion. Note that this number reflects the naive generation of trap properties—we simply generate one trap property for each structural element we aim to cover. Naturally, the desired coverage can typically be achieved with substantially fewer test cases—see discussion later in this section. Recall, however, that the aim of this case study was not to provide a minimal set of test cases providing the desired coverage, but instead to evaluate the scalability of using model checking techniques for test case generation—thus, we wanted to work with many properties to make our results representative of expected performance. Finally, note that each trap property for MC/DC coverage describes an *MC/DC pair* of test cases—we will have twice as many MC/DC test cases as we have trap properties.

Table 3 gives the performance figures in terms of time and memory of generating the suites to the three coverage criteria (a - in the table indicates that a run of the model checker was terminated after an excessively long run—more than 24 hours).

As the data in Table 3 illustrates, symbolic model checking does not seem to scale well beyond FGS03. For models FGS04 and FGS05, it quickly runs into problems. From Table 3 it is clear that memory usage is not the problem. To keep memory usage small, we are using the dynamic BDD variable reordering feature of NuSMV—without this option, NuSMV would exhaust the available memory quickly. Nevertheless, the dynamic variable reordering is quite costly and this deteriorates the performance of NuSMV to a point where the time to reorder becomes unbearable. In addition, the cost of constructing counterexamples in a symbolic model checker becomes a serious issue when the model checker is used for test case generation since we need a large number of counterexamples.

The bounded model checker, on the other hand, scales well to all FGS Models. To determine the search depth for the bounded model checker, we used results from a previous study using symbolic model checking for verification of the FGS system models [5]. In this previous study, we found that the full state space of FGS00 through FGS03 could be explored with 5 steps and with 12 steps in FGS04 and FGS05. Therefore, when generating state and table coverage, we used the default setting of 10 steps for FGS00 through FGS03 and 12 for FGS04 and 05. We attempted the same settings when generating MC/DC coverage, but the time required to search to this depth was simply unacceptable. Note here that the majority of the time was spent searching for test cases that are infeasible—a certain MC/DC pair did not exist. Searching to depth 12 for such non-existent test cases is counterproductive. Instead, we observed that the average test case length is quite short (Table 4 shows just a little over 2 for table coverage) and we simply set the search depth to a prudent 5. We expected this to assure that we found a large number of test cases, but did not waste any time searching for the ones that did not exist. Naturally, we may still miss some test cases that are longer than 5 should they exist (see discussion below).

As can be seen from the performance data for the bounded model checker in Table 3, even with a reduced search depth, the performance deteriorated quite notably when generating tests for MC/DC coverage (orders of magnitude slower than for the other coverage criteria). Two factors contribute to this phenomenon; (1) the length of the test sequences generated and (2) the complexity of the LTL properties to check.

Table 4 shows the average test case length we measured during our experiments. From the results it is clear that the test cases for MC/DC coverage were approximately twice as long as the ones for the other coverage criteria. Recall the short discussion on MC/DC in Section 6. The counterexample generated for an MC/DC trap property describes not one test case, but an *MC/DC test case pair*—the first test case takes a transition t out of state X with a particular truth as-

	State Coverage				Table Coverage				MCDC Coverage			
	Symbolic		Bounded		Symbolic		Bounded		Symbolic		Bounded	
	Time (s)	Memory (MB)	Time (s)	Memory (MB (depth))	Time (s)	Memory (MB)	Time (s)	Memory (MB (depth))	Time (s)	Memory (MB)	Time (s)	Memory (MB (depth))
FGS00	1.01	1	0.29	1 (10)	2.04	10	0.79	10 (10)	1.72	10	3.55	13 (5)
FGS01	4.58	11	0.61	11 (10)	7.3	15	2.14	14 (10)	6.72	15	11.4	19 (5)
FGS02	33.86	21	1.76	15 (10)	82.34	24	5.94	21 (10)	78.2	24	51.76	30 (5)
FGS03	251.37	27	3.74	19 (10)	469.91	33	11.99	29 (10)	520.95	32	137.34	49 (5)
FGS04	-	-	57.89	32 (12)	-	-	101.2	53 (12)	-	-	39167.8	110 (5)
FGS05	-	-	81.61	58 (12)	-	-	193.67	80 (12)	-	-	46196.91	165 (5)

Table 3. Execution Times and Memory Usage for all FGS Models

signment to the basic conditions, the second takes us back to state X but this time we have exactly *one* basic condition with a different truth assignment and we do not take transition t . Thus, the test case length is destined to be approximately twice the length of the test cases generated for the other criteria. The need for a deeper search dramatically decreases the performance of the bounded model checker.

In addition to the increased test case length, the LTL properties characterizing the test cases are significantly more complex for MC/DC coverage than for the other coverage criteria (again, see Section 6). The dramatically longer trap properties negatively affects the performance of the bounded model checker [24].

From this discussion we can conclude that a bounded model checker seems to be a suitable tool for test case generation from formal specifications; it scales well to systems of industrial relevance, it generates the shortest possible test cases, and it is fully automated. There are, however, some drawbacks. Most importantly, if the shortest test case needed to cover a specific feature in the model is longer than the search depth of the bounded model checker, we have no way of telling if the test case simply does not exist or if it is longer than the search depth. This is an issue particularly for MC/DC generation where there are a fair number of MC/DC pairs that simply do not exist—if the bounded model checker fails to find a test case, the determination if it indeed exists is now a manual process.

As mentioned previously, during the generation of the test suites we did not attempt to minimize the number of test cases to achieve a desired coverage. In fact, it is easy to see that our test case generation approach, where a test case is generated for each trap property, will lead to a large amount of duplicate coverage.

We performed a simple analysis to measure the level of duplication for the test suite generated to achieve table coverage—we simply executed the tests (sequentially) and kept track of the coverage achieved after each test. Most test

cases did not increase the coverage of the test suite—a clear indication that tests are redundant with respect to achieving coverage. Some tests, on the other hand, produced a “jump” in the coverage indicating that they exercise a new portion of the software. We created a reduced test suite using the tests that caused this jump in the coverage. Although this is not necessarily the minimal set to achieve the desired coverage, we found the difference in size between the initial set and this set to be such that it clearly indicated a large degree of duplication over the generated tests. For example we found that for FGS00 and FGS03 respectively, only 3 out of the 27, and, 11 out of 95 test cases were required to achieve table coverage. Since the cost of resetting a system and executing a new test case in some applications is high, identifying a small test suite that provides adequate coverage is of some importance. In future work we will investigate an iterative approach to the test case generation in order to achieve smaller test suites. Of course, a smaller test suite might achieve the same coverage, but it may reduce the defect detection capability of the test suite. We have just initiated a study to investigate how test suite size impacts the defect detection capability of the suite.

As mentioned in Section 2, we intended our test case generation framework to allow an analyst to generate test sequences from a formal specification and then run the tests on the implementation. Should additional tests be needed, we would like to generate the additional input sequences from the code. To this effect we evaluated an explicit state code model checker.

8. Java PathFinder Results

We have done some preliminary experiments on using the Java PathFinder (JPF) code-level model checker [25] to do test case generation on Java programs automatically generated from the RSML^{−e} models. Our initial motivation for

	State Coverage		Table Coverage		MCDC Coverage	
	Symbolic	Bounded	Symbolic	Bounded	Symbolic	Bounded
FGS00	2.8	1.5	5.6	2.2	9.0	3.5
FGS01	4.0	1.6	5.7	2.2	8.7	3.4
FGS02	5.3	1.5	6.3	2.2	9.1	3.9
FGS03	5.3	1.6	6.0	2.1	8.7	4.0
FGS04	-	1.7	-	2.2	-	3.9
FGS05	-	1.8	-	2.1	-	4.0

Table 4. The average length of the test cases generated.

using a code-level model checker was to investigate whether one can use such a tool to discover test cases for covering code that was not being covered by the test cases derived from the RSML^{-e} specification. Here however, since we are doing an automatic translation of RSML^{-e} to Java, we will use our preliminary results to judge how an explicit-state model checker (such as JPF) compares to the symbolic and bounded model checker approaches for test case generation for RSML^{-e} . We studied test cases for branch coverage at the Java level, since branch coverage is an often used code coverage criteria and, due to the translation used, corresponds closely to table (decision) coverage at the RSML^{-e} level.

The test case generation process using JPF is currently not automated, in particular, the trap properties are assertions added by hand, and, there is no facility to extract the test inputs from each counterexample produced. We therefore will not be reporting any specific timing and memory usage results, but rather make general observations. In short, the explicit-state model checker did not perform as well as the symbolic and bounded model checking approaches. For FGS00 the model checker could generate test cases to cover all branches within a matter of seconds while using an insignificant amount of memory (less than 1 Mb). Whereas for FGS03, it could not generate enough of the state-space to cover all the branches in a reasonable amount of time (3 hours). We did not attempt to generate tests for FGS04 or FGS05.

From the experiments it is clear that the explicit-state model checking is particularly sensitive to the length of the test cases required to achieve the desired coverage. For example, FGS03 has 10 boolean inputs at each input cycle (i.e., 2^{10} options), and the explicit-state model checker can at most deal with 3 such inputs (since the state space size will be $O(10^9)$).

Explicit-state model checking does however allow more control over the search, and we conjecture this can be exploited to do efficient test case generation. Specifically, one can use heuristic search [11] techniques to find the desired test cases—we will pursue this line of research in future work. Recently, the idea of combining symbolic execution with model checking to do test case generation has

been proposed [17]—this allows one to mitigate the effect of longer test cases and should therefore allow for more efficient test case generation. This latter approach is in some ways similar to doing bounded model checking, and we will investigate how these techniques compare in future work.

9. Summary and Conclusions

To summarize, we have conducted a series of case studies evaluating how well model checking techniques will scale when used for test case generation. Our experiences point out limitations of symbolic as well as explicit state model checkers. A bounded model checker, however, performed very well and shows tremendous promise. The domain of interest in our study has been safety critical reactive systems—systems that lend themselves to modeling with various formalisms based on finite state machines. In this domain, test cases providing common coverage seem to be quite short, thus making bounded model checkers perform very well.

Naturally, there are still many challenges to address. There are systems where the cost of restarting the system to execute a new test sequence is quite high. In this situation it is highly desirable to have long test cases that provides extensive coverage so that we can minimize the number of system restarts required to execute the test suite. The bounded model checking approach discussed performing well in our case study provides the exact opposite—we will get many very short test cases. Techniques to effectively merge these test cases to longer test sequences would be highly desirable. Alternatively, techniques based on explicit state model checking and heuristic searches may be able to provide long test cases that provides extensive coverage of a model. We plan to investigate this approach in the context of Java PathFinder shortly.

The nature of bounded model checking makes it unsuitable for verification. Determining the appropriate search depth to grantee that we find most (if not all) test cases without wasting time with deep searches for test cases do not exist remains a challenge.

Our case study example is non-typical in that it only con-

tains discrete variables, we have no integer or real variables in the model. Naturally, many models will have numerous numeric variables involved in various interrelated numeric constraints. Applying the model checking techniques on these systems will be a challenge. Recent advances bringing efficient decision procedures and bounded model checking together promises to help to some extent. Various abstraction techniques, for example, iterative refinement [14, 7] and domain reduction abstraction [6], also holds promise in this regard. We hope to conduct experiments on systems with these characteristics shortly.

References

- [1] P. E. Ammann and P. E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of the Fourth IEEE International Symposium on High-Assurance Systems Engineering*. IEEE Computer Society, Nov. 1999.
- [2] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, Nov. 1998.
- [3] J. Callahan, F. Schneider, and S. Easterbrook. Specification-based testing using model checking. In *Proceedings of the SPIN Workshop*, August 1996.
- [4] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200, September 1994.
- [5] Y. Choi and M. Heimdahl. Model checking RSML⁺-requirements. In *Proceedings of the 7th IEEE/IEICE International Symposium on High Assurance Systems Engineering*, October 2002.
- [6] Y. Choi, S. Rayadurgam, and M. Heimdahl. Automatic abstraction for model checking software systems with interrelated numeric constraints. In *Proceedings of the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE-9)*, pages 164–174, September 2001.
- [7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pages 154–169, July 2000.
- [8] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
- [9] A. Engels, L. M. G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *Proceedings of TACAS'97, LNCS 1217*, pages 384–398. Springer, 1997.
- [10] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, November 1999.
- [11] A. Groce and W. Visser. Model checking java programs using structural heuristics. In *Proceedings of ISSTA 2002*, Rome, Italy, July 2002.
- [12] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
- [13] M. P. Heimdahl, S. Rayadurgam, and W. Visser. *Proceedings of The First International Workshop on Automated Program Analysis, Testing and Verification, ICSE 2000*. 2000.
- [14] T. A. Henzinger, R. Jhala, R. Majumdar, and R. Majumdar. Lazy abstraction. In *Proceedings of the 29th Symposium on Principles of Programming Languages*, January 2002.
- [15] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *Proceedings of 2003 International Conference on Software Engineering*, Portland, Oregon, May 2003.
- [16] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '02)*, Grenoble, France, April 2002.
- [17] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Warsaw, Poland, April 2003.
- [18] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.
- [19] NuSMV: A New Symbolic Model Checking. Available at <http://http://nusmv.iirst.itc.it/>.
- [20] S. Rayadurgam and M. P. Heimdahl. Coverage based test-case generation using model checkers. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*, pages 83–91. IEEE Computer Society, April 2001.
- [21] S. Rayadurgam and M. P. Heimdahl. Test-Sequence Generation from Formal Requirement Models. In *Proceedings of the 6th IEEE International Symposium on the High Assurance Systems Engineering (HASE 2001)*, Boca Raton, Florida, October 2001. To appear.
- [22] J. M. Thompson, M. P. Heimdahl, and S. P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, number 1687 in LNCS, pages 163–179, September 1999.
- [23] J. M. Thompson, M. W. Whalen, and M. P. Heimdahl. Requirements capture and evaluation in NIMBUS: The light-control case study. *Journal of Universal Computer Science*, 6(7):731–757, July 2000.
- [24] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st Symposium on Logic in Computer Science*, pages pp. 322–331, Cambridge, June 1986.
- [25] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the 15th International Conference on Automated Software Engineering*, Grenoble, France, September 2000.